[09. 浏览器的那些事儿 —— 浅析浏览器的渲染原理](http://wiki.baidu.com/pages/viewpage.action?pageId=65662672)

[转至元数据结尾](http://wiki.baidu.com/pages/viewpage.action?pageId=65662672#page-metadata-end)

* 创建： [陈不骄](http://wiki.baidu.com/display/~deleted_chenbujiao)，最新修改于： [2015-03-09](http://wiki.baidu.com/pages/diffpagesbyversion.action?pageId=65662672&selectedPageVersions=3&selectedPageVersions=4)

[转至元数据起始](http://wiki.baidu.com/pages/viewpage.action?pageId=65662672#page-metadata-start)

**作者：鞠东昆**

        1990年至今，大家熟悉的浏览器已有25年历史，第一个网页浏览器WorldWideWeb（后更名叫Nexus）为互联网时代打开了门。众所周知，目前主流的浏览器有：IE、Chrome、Safari、Firefox，国内还有各种“套壳”的如Sogou、360、猎豹等多到几十个浏览器，种类虽多但浏览器的结构和工作原理大同小异，负责页面的渲染引擎在很长一段时间内也只有trident、gecko、webkit等数种，本文旨在讲述浏览器渲染引擎是怎样工作的，其主要结构以及常见的与渲染相关的问题，深入了解这些知识对从事WEB工作有很大的帮助。

1. 从地址栏输入URL，浏览器是怎样工作的

        简单的讲，在地址栏输入 https://www.baifubao.com 并回车会发生以下事情：

          •  浏览器引擎解析url向server发起网络请求；

          •  接收到server返回的数据包（超文本），交由渲染引擎解析并开始渲染；

          •  渲染的过程是按照标签（DOM树）的顺序进行的，遇到静态资源外链（如css、js、图片等），控制网络组件发起对这些资源的请求，注意这个过程浏览器是边渲染边请求/接收数据的。css、图片会立即被渲染引擎解析渲染（不会阻塞渲染），js会立即被JS引擎解析执行（会阻塞渲染，只有js执行完毕后才会继续渲染 ）；

          •  所有的外链资源请求完毕，页面渲染完毕/js执行完毕，一次完整的页面请求就完成了。

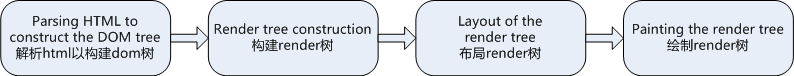
        看过上面的过程会发现浏览器是由分工明确的组件构成的，主要包括用户界面、浏览器引擎、渲染引擎、网络、用户界面后端、js引擎和数据存储等组件，从打开浏览器输入url到最终的页面出现，用户最关心的就是页面的载入速度和显示方式了，这与渲染引擎的工作方式息息相关，话不多说我们来看看何为渲染引擎，它是怎样渲染页面的，它为什么会直接影响页面的载入速度和显示方式。

2. 渲染引擎 (rendering engine)

        渲染引擎的职责就是渲染，即在浏览器窗口中显示所请求的内容。默认情况下，渲染引擎可以显示html、xml文档及图片，它也可以借助插件（一种浏览器扩展）显示其他类型数据，例如使用PDF阅读器插件，可以显示PDF格式的文件，flash插件可以显示flash动画等就不多说了， 这里只讨论渲染引擎最主要的用途—显示应用了CSS之后的html及图片。

2.1 渲染流程 (rendering flow)

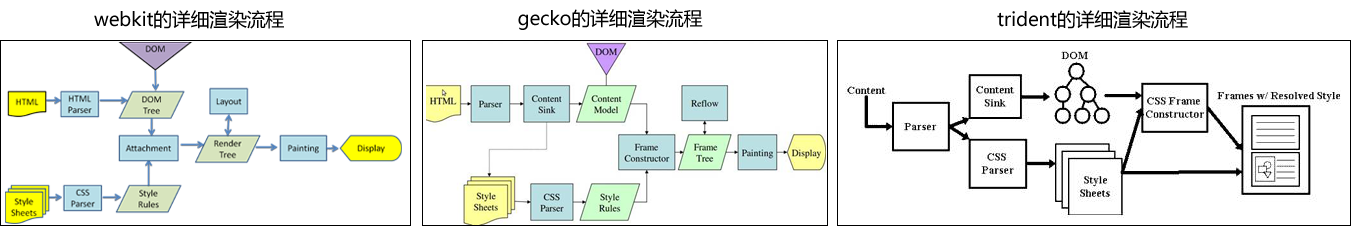
        常见的渲染引擎有trident (IE)、gecko (Firefox)、webkit (Chome&Safari)。虽说这些浏览器使用的渲染引擎不同，但他们的渲染引擎基本流程是相同的，在取得内容之后，渲染引擎是按照如下图所示的流程进行的。



        渲染引擎首先解析HTML文档，转换为一棵DOM树 (DOM tree) ，此为第一步。接下来不管是内联式，外联式还是嵌入式引入的CSS样式也会被解析，这些样式信息以及html中的可见性指令将被用来构建另一棵树---渲染树 (render tree)，渲染树包含带有颜色，尺寸等显示属性的矩形，这些矩形的顺序与显示顺序一致。然后就是对渲染树的每个节点进行布局处理，确定其在屏幕上的显示位置（确切的坐标）。最后就是遍历渲染树并用UI后端层将每一个节点绘制出来。

        值得注意的是，这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局渲染树。它是解析完一部分内容就显示一部分内容，同时可能还在通过网络下载其余内容。

        针对不同的渲染引擎具体步骤可能有所不同，就拿常见的webkit和gecko来说吧。



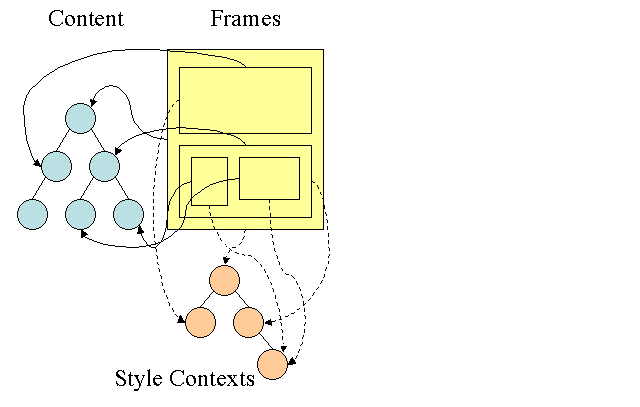
        尽管webkit、gecko和trident使用略微不同的术语，但这个过程还是基本相同的。gecko和trident里把格式化好的可视元素称做“帧树”(Frame tree)。每个元素就是一个帧 (Frame)。 webkit则使用“渲染树”这个术语，渲染树由“渲染对象”组成。webkit 里使用“Layout”表示元素的布局，gecko则称为“Reflow”。webkit使用“Attachment”来连接DOM节点与可视化信息以构建渲染树。一个非语义上的小差别是gecko和trident都在html与DOM树之间有一个附加的层 ，称作“content sink”，是创建DOM对象的工厂。

        上文说到了，浏览器在构造DOM树的同时也在构造着另一棵树---渲染树，与DOM树相对应暂且叫它Render树吧，我们知道DOM树为javascript提供了一些列的访问接口 (DOM API)，但这棵树是不对外的。它的主要作用就是把html按照一定的布局与样式显示出来，用到了CSS的相关知识。从MVC的角度来说，可以将render树看成是V，dom树看成是M，C则是具体的调度者，比如HTMLDocumentParser等。

        每一个渲染树的节点称之为renderer或者render object，查看webkit的源代码我们可以发现Renderer一个基础的类定义，这个类是所有renderer对象的基类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class RenderObject{      virtual void layout();      virtual void paint(PaintInfo);      virtual void rect repaintRect();      Node\* node;  //the DOM node      RenderStyle\* style;  // the computed style      RenderLayer\* containgLayer; //the containing z-index layer  } |

        从中我们可以发现renderer包含了一个dom对象以及为其计算好的样式规则，提供了布局以及显示方法。具体效果图如下：（gecko和trident的Frames对应renderers,content对应dom）



        具体显示的时候，每一个renderer体现了一个矩形区块的东西，即我们常说的css盒子模型的概念，它本身包含了一些几何学相关的属性，如宽度width，高度height，位置position等。每一个renderer还有一个很重要的属性，就是如何显示它，display。我们知道元素的display有很多种，常见的就有none、inline、block、inline-block，有关盒模型的碎碎念后文再详细阐述。

 2.2 解析 (analyze)

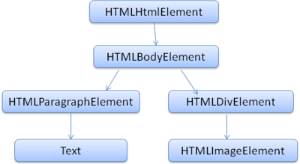
        解析一个文档意味着把它翻译成有意义的结构以供代码使用。解析的结果通常是一个表征文档的由节点组成的树，称为解析树或句法树。解析的过程涉及到语法分析、词法分析、转换等等这里就不多说了。

**2.2.1 html解析**

        html的词汇与句法定义在w3c组织创建的规范中，目前HTML5版本已经定稿，谈到规范就又扯远了，我们来看一个最简单的html解析例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <html>      <body>          <p>Hello World</p>          <div><img src="example.png" /></div>      </body>  </html> |

        会被转换成如下的DOM树：



        这里对分词算法和树的构建算法就不再多说，值得提到的一点是有些浏览器会对不完整的dom结构进行容错修复，而有些浏览器不会这么做。

**2.2.2 css解析**

        css用到的所有词汇定义规范如下：

|  |
| --- |
| comment     \/\\*[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*\/ num     [0-9]+|[0-9]\*"."[0-9]+  nonascii    [\200-\377]  nmstart     [\_a-z]|{nonascii}|{escape}  nmchar      [\_a-z0-9-]|{nonascii}|{escape}  name        {nmchar}+  ident       {nmstart}{nmchar}\* |

        注：ident代表样式中的class，name代表样式中的id。

　　css用到的语法BNF格式的定义如下：

|  |
| --- |
| ruleset   : selector [ ',' S\* selector ]\*      '{' S\* declaration [ ';' S\* declaration ]\* '}' S\*    ;  selector    : simple\_selector [ combinator selector | S+ [ combinator selector ] ]    ;  simple\_selector    : element\_name [ HASH | class | attrib | pseudo ]\*    | [ HASH | class | attrib | pseudo ]+    ;  class    : '.' IDENT    ;  element\_name    : IDENT | '\*'    ;  attrib    : '[' S\* IDENT S\* [ [ '=' | INCLUDES | DASHMATCH ] S\*      [ IDENT | STRING ] S\* ] ']'    ;  pseudo    : ':' [ IDENT | FUNCTION S\* [IDENT S\*] ')' ]    ; |

        每个HTML元素上，我们可能定义了很多不同类型的样式，如字体啦，颜色啦，布局啦等等。即使元素上不被我们定义样式，浏览器或者用户个性设置也会为它默认创造一些样式。

　　样式计算一项极其复杂的过程，我们定义样式的时候可以采用类似类的定义方式为一批元素设置样式，但是解析构造renderer的时候，浏览器是为每一个构造样式定义的。我们可能定义了极其多的样式而且有各种不同的规则，那找到元素匹配的样式规则是挺困难的。浏览器有多重算法错误来实现计算工作，具体就不细分析了，一个元素最终经过计算可能匹配到了很多条样式规则，他们之间存在一定的优先顺序，从低到高有：

1. 浏览器默认样式
2. 用户个性化浏览器设置
3. HTML开发者定义的一般样式
4. HTML开发者定义的!important样式
5. 用户个性化浏览器设置!important样式

2.3 布局 (Layout)

        上面确定了renderer的样式规则后，然后就是重要的显示因素布局了。当renderer构造出来并添加到render树上之后，它并没有位置跟大小信息，为它确定这些信息的过程，我们就称之为布局。HTML采用了一种流式布局的布局模型，从上到下，从左到右顺序布局，布局的起点是从render树的根节点开始的，对应dom树的document节点，其初始位置为 (0,0)，详细的布局过程为：每个renderer的宽度由父节点的renderer确定。父节点遍历子节点，确定子节点的位置 (x,y)，调用子节点的layout方法确定其高度。父节点根据子节点的height,margin,padding确定自身的自身的高度。

        为了避免因为局部小范围的DOM修改或者样式改变引起整个页面整体的布局重新构造，浏览器采用了一种dirty bit system的技术，使其尽可能的只改变元素本身或者包含的子元素的布局。当然有些情况无可避免的要重新构造整个页面的布局，如适合于整体的样式的改变影响了所有renderer，如body{font-size:111px} 字体大小发生了改变，还有一种情况就是浏览器窗口进行了调整，resize。

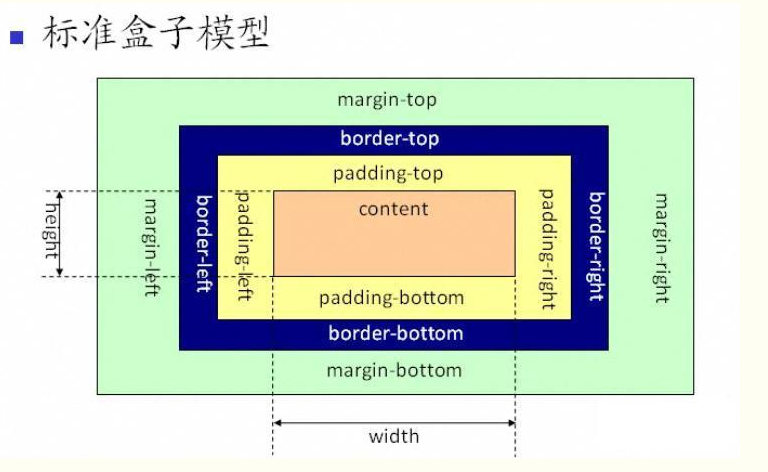
2.4 盒模型 (The CSS Box Model)

        网页设计中常听的属性名：内容 (content)、填充 (padding)、边框 (border)、边界 (margin)， CSS盒子模式都具备这些属性。这些属性我们可以把它转移到我们日常生活中的盒子（箱子）上来理解，日常生活中所见的盒子也就是能装东西的一种箱子，也具有这些属性，所以叫它盒子模式。

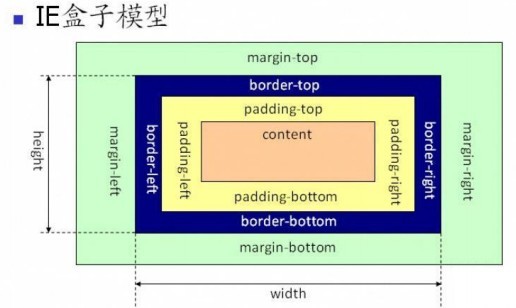
        每个网页的元素组成都是盒模型的，如我们的钱包站点就是有大大小小的很多个盒子构成：



        标准的盒子模型应该是这样的：



        但我们发现，不同浏览器对盒子模型的定义是有微小差别的，有些浏览器并没有遵循标准盒子模型的规范，比如IE浏览器，它的盒模型定义是这样的：



        眼神儿好的你就会发现IE浏览器实现的盒模型，在获取dom节点的宽高是与标准盒模型的不同，其将padding与border也一同算在内了，在不同的浏览器中像这种不遵循标准的例子还有很多，比如不同内核的浏览器对html标签定义了不同的默认样式（文字大小颜色、宽高、行距等），这会导致同样的代码在各个的浏览器表现是不同的，这就是所谓的兼容性问题。

        为解决这些兼容性问题（浏览器对文档渲染的差异），我们需要用某些技术手段尽量使得各个浏览器能够表现的如我们预期的那样，比如上面提到的html标签默认属性不同的问题，可以通过使用我们认为标准的css样式强制将这些html标签重新定义，以达到各个浏览器对html标签的默认样式是相同的。

3. 总结

        随着技术的不断更新，浏览器也在不停的升级和创新，为追求更好的性能和用户体验，针对页面渲染的技术方案也层出不穷，比如最近比较火的React-Canvas就是利用html5中的canvas标签（一个可以在页面上绘图的标签）来模拟浏览器渲染引擎所做的事情，这样做的好处在于，只用一个 Canvas DOM 元素，降低 DOM 数量与渲染的复杂度，渲染部分大部分交由GPU处理，而控制运算（逻辑）则可以用 JavaScript 在 CPU 里做，这样可以充分利用硬件资源，这样的页面在移动端的体验是非常极致的。这些页面渲染的创新与技术革命其实也都没有离开本文讲述的渲染引擎原理，可见往深处走一步，看看浏览器底层的工作原理是受益匪浅的。